Science of Computer Programming 106 (2015) 78–92

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Correctness issues on MARTE/CCSL constraints

Frédéric Mallet [a,b,c,d,*], Robert de Simone [b,d]

[a] Université Nice Sophia Antipolis, France
[b] INRIA Sophia Antipolis Méditerranée, France
[c] East China Normal University, Software Engineering Institute, China
[d] I3S Laboratory, UMR 7271 CNRS, France

ARTICLE INFO

Article history:
Received 6 April 2013
Received in revised form 24 February 2015
Accepted 2 March 2015
Available online 5 March 2015

Keywords:
Logical time
Architecture-driven analysis
UML MARTE
Reachability analysis

ABSTRACT

The UML Profile for Modeling and Analysis of Real-Time and Embedded systems promises a general modeling framework to design and analyze systems. Lots of works have been published on the modeling capabilities offered by MARTE, much less on available verification techniques. The Clock Constraint Specification Language (CCSL), first introduced as a companion language for MARTE, was devised to offer a formal support to conduct causal and temporal analysis on MARTE models.

This work relies on a state-based semantics for CCSL to establish correctness properties on MARTE/CCSL specifications. We propose and compare two different techniques to build the state-space of a specification. One is an extension of some previous work and is based on extended finite state machines. It relies on integer linear programming to solve the constraints and reduce the state-space. The other one is based on an intentional representation and uses pure Boolean abstractions but offers no guarantee to terminate when the specification is not safe.

The approach is illustrated on one simple example where the architecture plays an important role. We describe a process where the logical description of the application is progressively refined to take into account the execution platform through allocation.

1. Introduction

The Unified Modeling Language (UML 2.x) proposes a simplistic and informal model of time, called *Simple Time*. This model has been extended in the UML Profile for Modeling and Analysis of Real-Time and Embedded systems [1] (MARTE), adopted in November 2009. MARTE introduces a richer *Time model* [2] general enough to support different forms of time (discrete or dense, chronometric or logical). Its so-called *clocks* allow enforcing as well as observing the occurrences of events and the behavior of annotated UML elements. The *Clock Constraint Specification Language* (CCSL) has been initially defined in an annex of the MARTE specification to provide a concrete syntax for handling these logical clocks as first-class citizens. It was endowed with a formal operational semantics [3] to breathe life into UML models by defining synchronization and coordination schemes between the various modeling elements.

The operational semantics of CCSL is adequate to build a simulation framework, like TimeSquare [4][1] but less appropriate to conduct exhaustive analyzes. We rely for that purpose on a state-based semantics to establish correctness properties on

* Corresponding author at: Université Nice Sophia Antipolis, France.
E-mail address: Frederic.Mallet@unice.fr (F. Mallet).

[1] http://timesquare.inria.fr.

http://dx.doi.org/10.1016/j.scico.2015.03.001
0167-6423/

CCSL specifications. A CCSL specification is called *safe* if and only if it can be represented with a finite state machine. Some of the CCSL constraints are not safe and their semantics can only be captured with an infinite number of states or a finite symbolic representation of these infinite states. In a previous work [5], we have proposed to use extended state machines, *i.e.,* finite state machines extended with (unbounded) integer variables, to capture the semantics of unsafe constraints. This abstraction leads to the generation of observers for simulation or model-checking verification. This abstraction is very convenient since it does not need to assume that the specification is actually safe. In [6] we have proposed an algorithm to detect safe specifications. Having that we propose an alternative solution that does not rely on extended finite state machines but rather on an *intentional* data structure. In this paper, we propose an extension of [5] with state invariants to further reduce the size of the product. Then, we describe the alternative solution. Both solutions have advantages and flaws that are explored in details.

Building the synchronized product of a CCSL specification is key to conduct model-checking on properties. Indeed, we also discuss some classical liveness issues that may arise with CCSL specifications and that can actually be checked with our proposal. Safety and liveness issues are illustrated on a simple example borrowed from AADL and in which the platform, the architecture and the binding are captured in MARTE/CCSL.

Section 2 starts with a positioning with respect to related works. Section 3 gives some background about CCSL and transition systems for clock systems. Section 4 is the main part of the contribution. It discusses first the solution relying on extended finite state machines and integer linear programming. Second, it compares it to another solution relying on purely Boolean analysis and using an intentional data structure. Section 5 gives an illustrative example and discusses typical correctness properties.

2. Related work

Logical clocks have been introduced in distributed systems [7] to loosely synchronize communicating systems and order their events. This appealing concept of logical clock is central in many contexts and for several purposes, including in process networks or in Petri nets [8], however its usage in CCSL is mainly inspired from their central role as activation conditions in Synchronous languages [9,10]. Compared to Lamport's clocks, synchronous languages introduce the notion of atomic reaction (also called *instant*) in which several events occur simultaneously. Consequently, the behavior of a system can be defined based on what has happened during the reaction, but also on what has NOT happened, hence leading to the so-called reaction to absence. CCSL operators forbid the occurrence of some events based on what has happened or not in previous reactions.

Whereas usually in synchronous languages, the programmer handles signals (sequences of values) as a primitive construct, the notion of signals does not exist in CCSL and the language only focuses on the clocks themselves. In synchronous languages, clocks tend to be handled mainly by the compiler, through the process called *clock calculus*, to decide when the values are valid and when the computations must be performed. In Lustre [11], clocks of inputs are usually given indirectly while in Esterel [9] they are rarely handled explicitly by the programmer. In polychronous extensions, like Signal [12], the clocks constrain the system to become endochronous, when the system is underspecified. CCSL was devised as a language focusing on clocks independently of signals and values.

Later, the notion of tag system [13] and then tag structure [14] was proposed as a mathematical framework to compare models of computations and orchestrate heterogeneous models. CCSL provides a concrete syntax [15] for building such orchestration models by focusing only on clocks (or tags) and not on values.

Initially the operational semantics of CCSL was given as a set of rewriting rules [3] in order to build a simulation engine that performs the clock calculus dynamically on the fly. To conduct exhaustive analyzes on CCSL specifications we propose to encode the semantics using transition systems. Some CCSL operators cannot be represented with finite transition systems and symbolic representations must be proposed to deal with these so-called *unsafe* operators. In [5], finite state machines extended with unbounded integer variables were proposed for that purpose. Integer variables symbolically captured the infinite number of states. We consider in this paper an alternative encoding that maintains an intentional representation of the infinite transition systems and that expands them on-demand when the synchronized product is built. The contribution of this paper is to compare the two alternative approaches and to discuss solutions to establish correctness properties on CCSL specifications.

Clearly, the proposed structure comes close to pushdown automata. The literature is abundant on pushdown automata (PDA). The class needed here is strictly weaker than PDA since the operations on the stack are very limited. Indeed, the stack would just be used for counting ($+1$, -1, zero-test). Counter automata [16,17] appears to be a subclass closer to our needs. The reachability problem for counter-automata has been studied a lot and subclasses for which reachability is decidable have been identified [18,19]. A naive encoding of CCSL in counter automata would probably result in having one counter per clock or at least one counter per clock domain. Such an encoding is out of the scope of this paper. Relying on acceleration techniques [20] to compute the reachability set from a CCSL specification is clearly an interesting problem that is also beyond the goals of this contribution. This paper does not propose a new mathematical abstraction but rather explores two practical solutions to conduct exhaustive verifications on CCSL specifications.

Several attempts have been made before to perform exhaustive analyses of CCSL specifications. Gaston et al. [21] proposed an encoding as Büchi automata to compare the expressiveness of CCSL with temporal logics. Yin et al. [22] proposed to encode CCSL operators in Promela to perform model-checking with the SPIN model-checker. In both attempts, only a safe